Octave Gaspard

May 5, 2024

CSE306

Raytracer report

1 Introduction

1.1 Using the raytracer

Use *make build* to build the *./render.exe* exectuable.

Run *./render.exe help* to know more about command line arguments.

Most camera settings are not fixed during compilation and may be modified as arguments, but the other parameters are to be modified in the code.

In the following cases, the code needs to be directly altered:

1. Modifying or adding elements to the scene, spheres, meshes or light sources (directly in main function, currently around line 1150)
2. The previous point includes the addition of movement functions, procedural textures, etc.
3. Modifying the value of gamma correction (currently at line 87)
4. Modifying the number of threads on which we run the rendering (currently at line 1176)

Spheres are built using the following constructor: (Vector Origin, double Radius, Vector Albedo, double Refraction (defaults to -1 as a diffuse surface, 0 for a mirror), Vector (*MovementFunc)(double) (a pointer to a Vector function taking values in [0, 1] for motion blur, defaults to &constant_position), Procedural* Texture (defaults to nullptr to use no procedural texture))

TriangleMeshes are similar: (const char* objFileName, const char* uvFileName, Vector Origin, double rescalingFactor, Vector (*MovementFunc)(double), Procedural* Texture, bool is_mirror (defaults to false))

1.2 Report

The plan the report follows is the same as the TDs. There will be examples for every feature and the implemented bonus ones for each TD, as well as an additional section at the end for extra features that were not even marked as bonus.

Notice some features were not deactivated to render the examples of the first TDs, so the examples may not be the most accurate of what we are trying to depict in each section.

All render times are on a laptop using 32 threads, running on a Ryzen 5500U.

2 TD1 features

2.1 Diffuse surfaces, direct lighting, shadows, mirrors, refraction

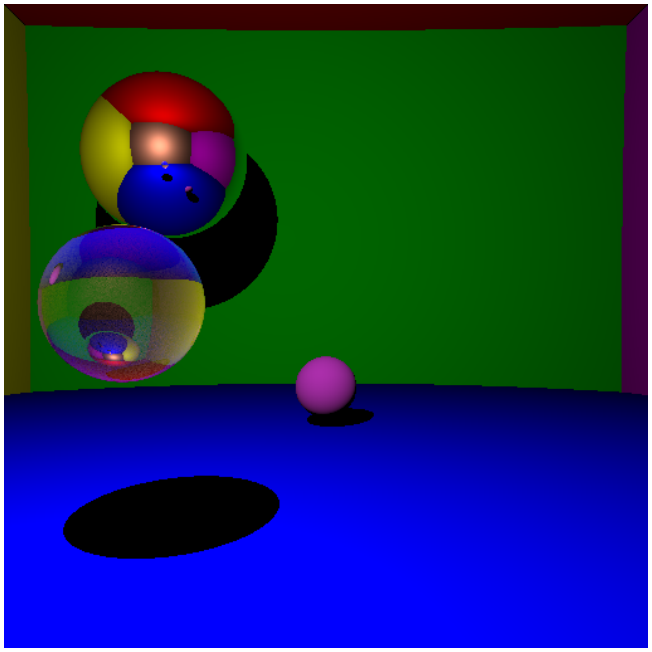

Figure 1: Example render in 512x512 of direct lighting for TD1 components, 128 monte-carlo samples. Refractive index of 1.49 for the lens. Rendered in a total of 4.54s.

Notice that in Figure 1 we have some reflection in the lens. This is because of the Fresnel effect, which is noisy because of the few monte-carlo samples used to render this image.

2.2 Fresnel for different refractive index

Notice in Figure 2 that with increasingly high refraction we also get inscreasingly high reflection, and the added monte-carlo samples

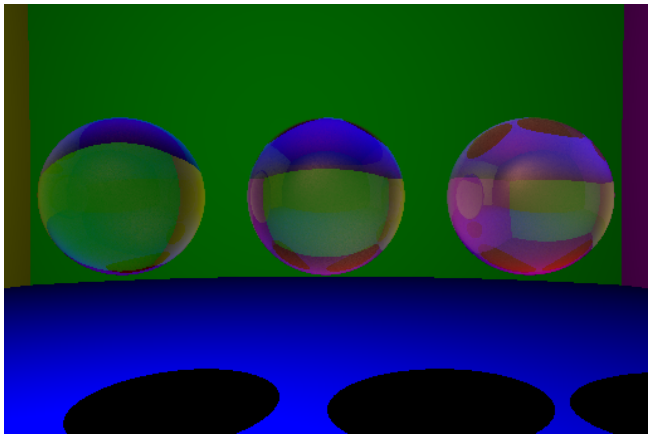

Figure 2: 512x342 render showing the Fresnel effect, 1024 monte-carlo samples. Refractive index from left to right: 1.2, 1.49, 2. Rendered in 25.75s.

3 TD2 features

3.1 Indirect lighting

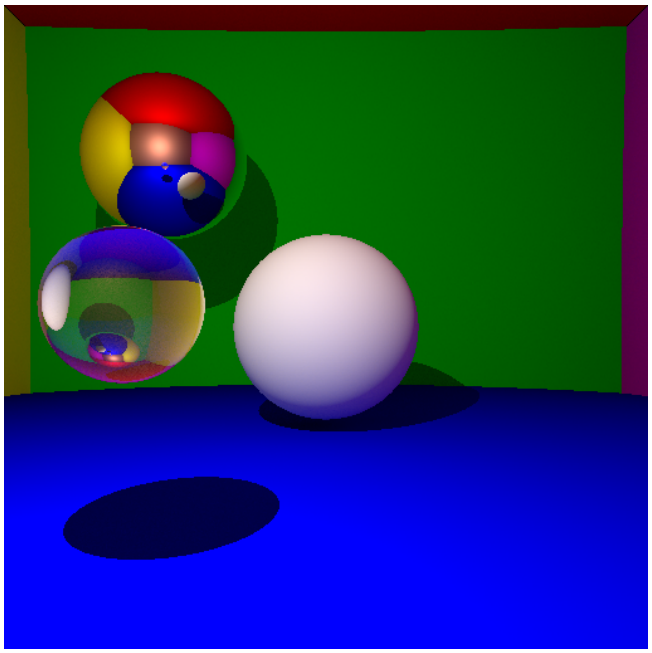

Figure 3: 512x512 render with 1 bounce of indirect lighting, 256 monte-carlo samples. Rendered in 17.34s.

Notice in Figure 3 how the white ball changes color based on the color of the surrounding balls, and how the shadows are not pitch black anymore.

Going from 1 to 5 light bounces make the colors brought by indirect lighting more vivid as seen in Figure 4 but also strongly increase rendering times.

Notice we used jittered sampling to get the first light bounce of each ray sent during monte-carlo integration.

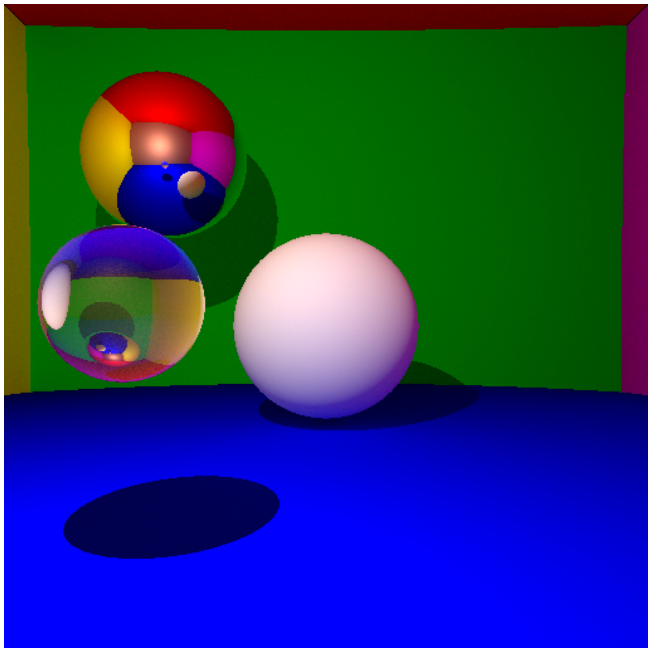

Figure 4: 512x512 render with 5 bounces of indirect lighting, 256 monte-carlo samples. Rendered in 48.8s.

3.2 Antialiasing

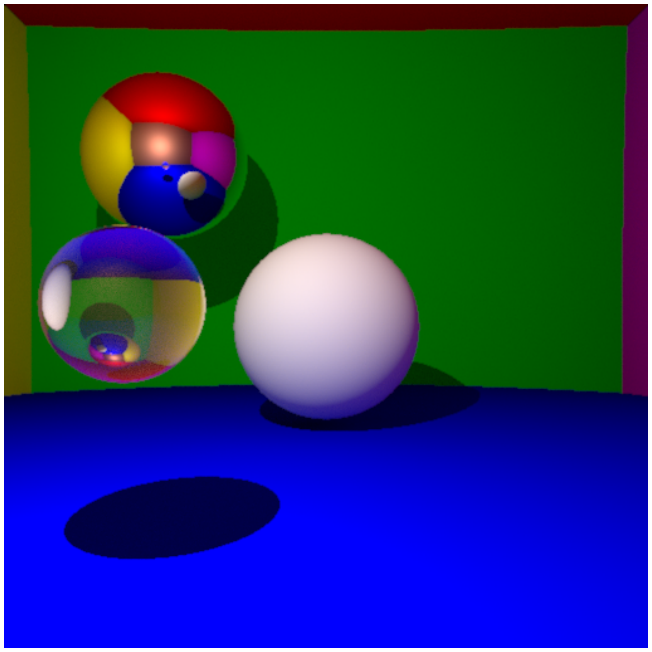

Figure 5: 512x512 render similar to Figure 3 with antialiasing strength 0.4. Rendered in 16s.

Notice Figure 5 has a reasonable amount of antialiasing leading in a reduction of sharp edges between pixels, and Figure 6 uses stronger antialiasing, causing it to also blur the image which may be an unwanted effect.

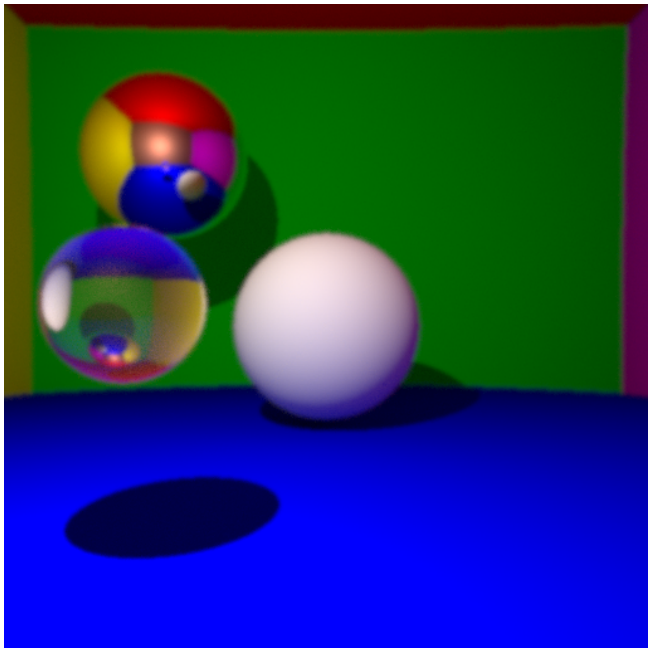

Figure 6: 512x512 render similar to Figure 3 with antialiasing strength 1.5. Rendered in 16.14s.

4 TD3 and TD4 mesh support

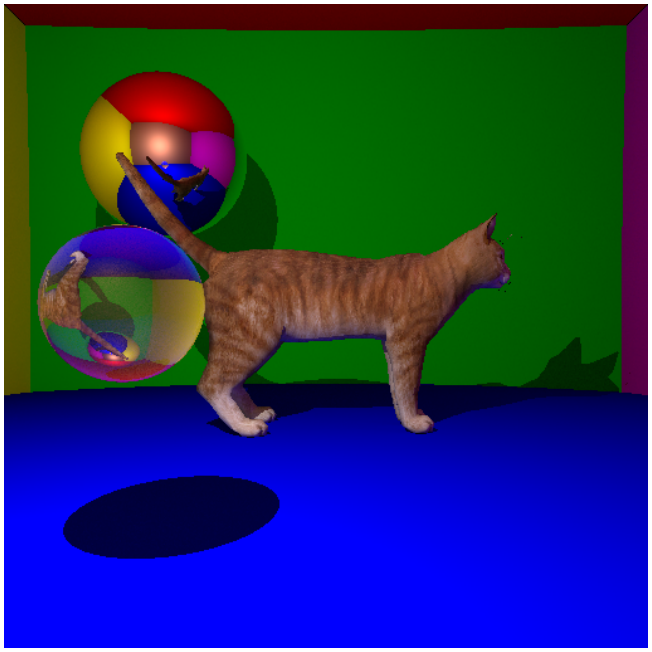

Figure 7: 512x512 with 1 bounce of indirect lighting and 256 monte-carlo samples. Mesh rescaled at 60%. Rendered in 28.37s.

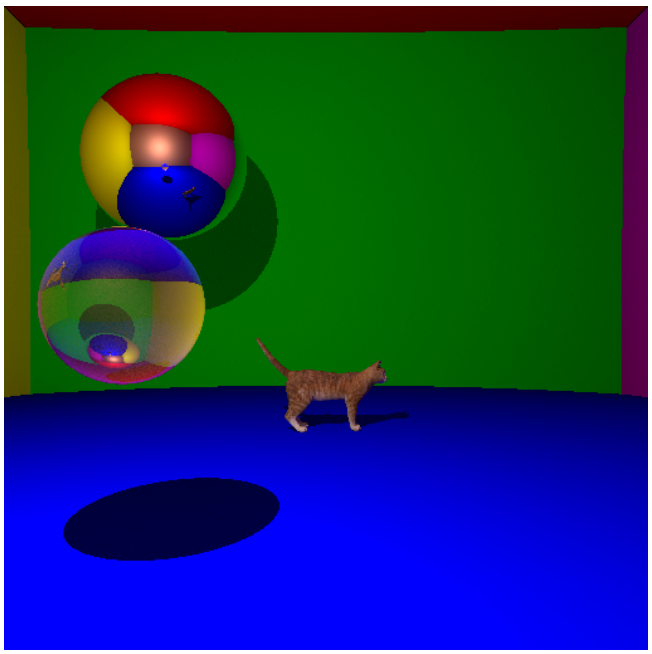

Figure 8: 512x512 with 1 bounce of indirect lighting and 256 monte-carlo samples. Mesh rescaled at 20%. Rendered in 25.1s.

Trying to intersect with every mesh of the cat would be very computationally expensive. Instead, we used the bounding box technique to reduce costs. This is illustrated by the 3 extra seconds needed to render Figure 7 with a large cat compared to Figure 8 with a smaller cat.

This additional time would actually be much larger if we did not also implement bounding volumes hierarchy. In addition, we also traverse the bounding box tree using a DFS to not try to intersect with boxes that are further than the current best intersection candidate found.

5 Additional features and final example render

Another reason causing the render in Figure 8 to be much faster is because we implemented bounding volume hierarchies, not with the middle point heuristic, cutting each box in two along its longest axis, but we implemented the Surface Area Heuristic following the algorithm detailed in this section of [1], and so even the rendering of the large cat is done in a very efficient way.

We also implemented Depth of field, general Motion Blur and a framework for Procedural textures as well as Perlin noise.

We show in Figure 9 an image demonstrating all these tools. We notice a cat and a sphere at the bottom of the image have Perlin noise textures. We also notice the two spheres on the left, closer to the camera, are blurry due to the Depth of Field settings. Finally, the sphere in the top right seems to be falling after being launched, due to the movement function used for Motion Blur.

Notice this render also hase 2 point lights instead of one, but the render is not slowed down since we randomly pick only one light to test for direct lighting during computation, using as heuristic the intensity the light would give if no object was in

Figure 9: 512x512 render 1 bounce of indirect lighting, 256 monte-carlo samples, DoF distance 55 and radius 0.5, antialiasing strength 0.7. Rendered in 41.7s

6 Classes and thread organization

In addition to the Light struct and the Geometry class discussed in the lecture notes, we used a virtual Procedural class to be able to add any procedural texture in addition to the implemented Perlin noise.

We also put the bounding boxes inside a class, so that their logic is separate from the details of manipulating meshes, and could be extended to also hold Spheres within them.

Each thread is run on a subset of the lines of the pictures (each one is passed a block), and constructs its own random number generator, using a thread-safe method inspired from this post.

Since some parts of the picture may contain more complex geometry, some threads will be longer to terminate than others. However, if we take enough threads (32 by default in our case), this difference in work should be well balanced by the scheduling of the operations, since even if some threads finish early, there will still be enough left to cover every idle core of the CPU.

Then, the renderer first divides lines between threads, then threads call get_color() for each pixel. This function handles the logic of monte-carlo integration as well as antialiasing and depth of field, and gets the color of a single ray from a call to get_color_aux(), which itself holds the logic for mirrors, refraction, fresnel, direct and indirect lighting, delegating intersections to scene_intersect() itself using the intersect() method of Geometry objects.

References

[1] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016.